Volker Roth

Rechnersicherheit, SoSe 21

Übung 01

TutorIn: Oliver Wiese
Tutorium 02
Materialien: Latex, VSC, Skript

6. Mai 2021

---

1 (Un-)salted password hashes

We used the simple brute force method and tried all possible password variations. With the given alphabet which contain 26 characters, this took your program $62^4 = 14.776.336$ many iterations.

```
#alphabet
string_alphabet  = string.ascii_lowercase + string.ascii_uppercase + '0123456789'
alphabet = list(string_alphabet)

#create dictonary of file
dict = {}
seed = ''
with open("sha2pwd.txt", "r") as file:
    #strip unnecessary leading and ending chars
    seed = file.readline().lstrip("Seed: ").rstrip("\n")
    for line in file:
        hash = line.rstrip("\n")
        dict[hash] = 0

#password test each possible combination of alphabet with 4 chars
for char1 in alphabet:
    for char2 in alphabet:
        for char3 in alphabet:
            for char4 in alphabet:
                password = char1 + char2 + char3 + char4
                salt = seed+password
                hash = hashlib.sha256(salt.encode()).hexdigest()
                try:
                    test = dict[hash]
                    dict[hash] = password
                    #print(hash, password)
                except:
                    pass

#result to file
with open('result_single_process.txt', 'w') as file:
     file.write(json.dumps(dict))
```

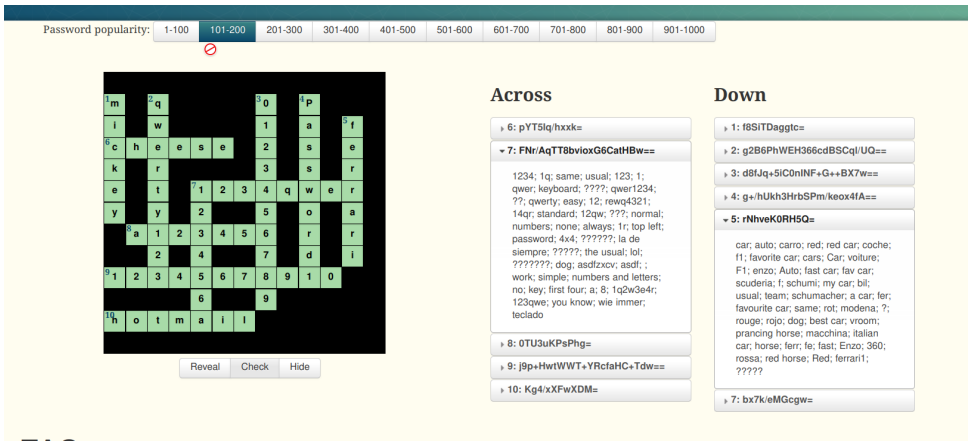
Moreover the also implemented the idea above with multithreading and multiprocessing.
As expected the implementation with multithreading changed nothing in terms of execution time.
The multiprocessing implementation however gave us quite a boost.
Below is a snippet of the multiprocessing code.

```python
def run(tid, task, seed, hash_dict):
    #password test each possible combination of alphabet with 4 chars
    for char1 in task:
        for char2 in alphabet:
            for char3 in alphabet:
                for char4 in alphabet:
                    password = char1 + char2 + char3 + char4
                    salt = seed+password
                    hash = hashlib.sha256(salt.encode()).hexdigest()
                    try:
                        test = hash_dict[hash]
                        #save password to file
                        with lock:
                            with open('result_multi_processing.txt', 'a') as file:
                                file.write(hash + ': ' + password +'\n')
                    except:
                        pass

def main():
    #split into qeual tasks of size TASK_SIZE for processes
    tasks = []
    single_task = []
    for char in string_alphabet:
        single_task.append(char)
        if len(single_task) == TASK_SIZE:
            tasks.append(single_task)
            single_task = []
    if single_task != []:
        tasks[-1] = tasks[-1] + single_task

    #delete file if it exists else create it
    with open('result_multi_processing.txt', 'w') as file:
        pass

    #start processes
    process_list = []
    for i in range(len(tasks)):
        process = Process(target=run, args=(i, tasks[i], seed, hash_dict))
        process.start()
        process_list.append(process)

    #wait for all processes to finish
    for process in process_list:
        process.join()
```

```
[redacted]:~/Dokumente/rechnersicherheit/rechnersicherheit-sose-21/u2$ python3 password_multi_processing.py
enter number of processes: 1
--- 13.36160659790039 seconds ---
[redacted]:~/Dokumente/rechnersicherheit/rechnersicherheit-sose-21/u2$ python3 password_multi_processing.py
enter number of processes: 2
--- 6.8104634284973145 seconds ---
[redacted]~/Dokumente/rechnersicherheit/rechnersicherheit-sose-21/u2$ python3 password_multi_processing.py
enter number of processes: 4
--- 3.667797088623047 seconds ---
[redacted]~/Dokumente/rechnersicherheit/rechnersicherheit-sose-21/u2$ python3 password_multi_processing.py
enter number of processes: 16
--- 1.8293890953063965 seconds ---
[redacted]:~/Dokumente/rechnersicherheit/rechnersicherheit-sose-21/u2$
```

2 Password breaches

a)



b)

Passwords are not salted, so duplicates stand out.
Even worse, the use of the block mode encrypts blocks of bytes separately, so you can see duplicate parts of the passwords. You can see this in the comic, for example the later half of the passwords with hint „Best TOS Episode“ and „sugarland“ are the same, and the first half of the passwords with hints „Best TOS Episode“ and „sexy earlobes“ are the same. That’s why the comics calls it a crossword puzzle.
Password hints should maybe also be encrypted?

3 Project

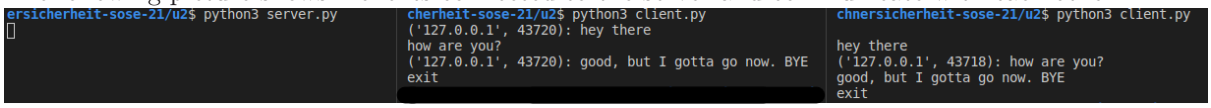
In the following you will see a snippet of python code from the server side.
We used threads and sockets for the task

```python
    def run(self):
        #incoming msg thread
        if self.tid == 0:
            #while client_socket is alive
            while not self.client_socket._closed:
                #recv msg from client fo size DATASIZE
                msg = self.client_socket.recv(DATASIZE)
                #if client disconnected
                if msg == bytes(0):
                    #remove client from connected_clients and exit
                    with lock:
                        connected_clients.remove(self.client_socket)
                        #write disconnect to log
                        with open('log.txt', 'a') as file:
                            date = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
                            file.write(date + ': ' + str(self.client_adress) + ' disconnected\n')
                    #stop thread
                    sys.exit()
                #store msg in msg_buffer
                with lock:
                    msg_buffer.append(myMsg(msg, self.client_socket, self.client_adress))

        #outgoing msg thread
        if self.tid == 1:
            while True:
                time.sleep(1)
                #if msg to send not empty and clients are available
                if msg_buffer != [] and connected_clients != []:
                    #send msg to each client
                    for s in connected_clients:
                        #if client is still connected and client is not the sender of the msg
                        if not s._closed and s != msg_buffer[0].socket:
                            #send msg to client
                            s.sendall(msg_buffer[0].get_final_msg_bytes())
                    #remove msg from bugger
                    with lock:
                        mymsg = msg_buffer.pop(0)
                        #write msg to log
                        with open('log.txt', 'a') as file:
                            date = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
                            file.write(date + ': ' + 'msg send: "' + mymsg.get_final_msg_bytes().decode('utf-8') + '" \n')
```

Here is the snippet from the client side.

```python
    def run(self):
        #incoming msg thread
        if self.tid == 0:
            #while server_socket is alive
            while not self.server_socket._closed:
                #waits for msg from server of size DATASIZE
                msg = self.server_socket.recv(DATASIZE)
                #print msg
                print(msg.decode('utf-8'))

        #outgoing msg thread
        if self.tid == 1:
            #while server_socket is alive
            while not self.server_socket._closed:
                #wait for keyboard input
                msg = input()
                if msg == 'exit':
                    os._exit(1)
                #send msg
                server_socket.send(bytes(msg, 'utf-8'))
```

The following picture shows 2 clients connected to the server and communicate with each other.



This is the log file produced by the server.

```
05/05/2021 10:26:38: new connection from('127.0.0.1', 43724)
05/05/2021 10:26:48: new connection from('127.0.0.1', 43726)
05/05/2021 10:26:54: msg send: "('127.0.0.1', 43724): hey there"
05/05/2021 10:27:02: msg send: "('127.0.0.1', 43726): how are you?"
05/05/2021 10:27:17: msg send: "('127.0.0.1', 43724): good, but I gotta go now. BYE"
05/05/2021 10:27:18: ('127.0.0.1', 43724) disconnected
05/05/2021 10:27:21: ('127.0.0.1', 43726) disconnected
```